Matias Kangasjärvelä

Vulkanin Perusteet

Insinööri (AMK),

tietotekniikka

Kevät 2017

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tekijä: Matias Kangasjärvelä

Työn nimi: Vulkanin perusteet

Tutkintonimike: Insinööri (AMK), tietotekniikka

Asiasanat: Vulkan, 3D Grafiikka, C++, Grafiikkarajapinta

Opinnäytetyön tavoitteena oli luoda käyttöohje Vulkanin perusteisiin ja eroihin muihin grafiikkarajapintoihin verrattuna. Käyttöohjeen seuraaminen vaatii Windows-käyttöjärjestelmän sekä Visual Studio 2015 tai uudemman kehitysympäristön. Lisäksi Vulkanin käyttäminen vaatii tuen näytönohjaimen ajureilta ja Vulkan ohjelmistokehityspaketin.

Grafiikkarajapinnalla tarkoitetaan ohjelmointirajapintaa, joka mahdollistaa 3D-grafiikan tuottamisen näytönohjaimella. Vulkan on niin sanottu moderni grafiikkarajapinta, joka mahdollistaa paremmin näytönohjaimien resurssien optimaalista hyödyntämistä.

Ensimmäinen osa työstä käy läpi grafiikkarajapintoja yleisesti. Toinen osa syventyy Vulkaniin yleisellä tasolla ja käy läpi sen historiaa ja eroja muihin grafiikkarajapintoihin. Lopuksi kolmannessa osassa työtä käydään läpi yksinkertaisen Vulkan ohjelman avulla Vulkanin perusteita.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Author: Matias Kangasjärvelä

Title of the Publication: Basics of Vulkan

Degree Title: Bachelor of Engineering

Keywords: Vulkan, 3D Rendering, C++, Graphics API

The aim of the thesis was to create a guide into the basics of Vulkan and compare it to other graphics APIs (application programming interface). Graphics API is an application programming interface that makes it possible to produce 3D-graphics with the graphics card. Vulkan is a so called modern graphics API that enables the program to make more optimal use of the resources available in the graphics card.

Following the guide requires use of the Windows operating system and the Visual Studio 2015 or newer integrated development environment. In addition to this the use of Vulkan requires support from the driver of the graphics card and the Vulkan software development kit.

First part of the thesis goes over the different graphics APIs on a general level. The second part focuses on Vulkan on a general level and goes through the history Vulkan and differences compared to other graphics APIs. Lastly the third part goes over the basics of Vulkan using a simple Vulkan program.

The final result of this thesis was a look into the basics of Vulkan and some concepts of 3D-graphics. The tutorial could be used by any C++ programmer trying to create applications with Vulkan.

SISÄLLYS

TERMILUETTELO

AMD	Advanced Micro Devices, Inc. Yksi suurimpia näytönohjaimien ja prosessoreiden valmistajia.
DICE	Digital Illusions CE AB. Ruotsalainen mm. Battlefield ja Mirror's Edge pelisarjoista tunnettu pelistudio.
GLFW	Ohjelmointikirjasto, joka mahdollistaa mm. ikkunoiden luomisen ja käyttäjän syötteen hallitsemisen käyttöjärjestelmä riippumattomasti.
GLSL	OpenGL Shading Language. Näytönohjaimen varjostimien kirjoittamiseen käytetty ohjelmointikieli.
Khronos Group	Vulkania ja OpenGL kehittävä konsortio.
LunarG	Vulkanin ohjelmistokehityspaketin kehityksestä ja levityksestä vastaava yritys.
Verteksi	3D-grafiikassa käytettyjen kolmioiden kärkipiste.
SPIR-V	Standard Portable Intermediate Representation versio V. Vulkanin käyttämä näytönohjaimen varjostimien välikieli, johon toisessa kielessä kirjoitettu varjostin tulee kääntää.

1 JOHDANTO

3D-grafiikan käytön yleistyessä 90-luvun alkupuolella alkoivat yleistyä myös näytönohjaimet, jotka mahdollistavat 3D-grafiikan piirtämisen. Monet näistä näytönohjaimista toimivat eri tavoin ja ohjelmien tuli lisätä tuki näytönohjaimille yksitellen. Tähän ongelmaan ratkaisuksi kehitettiin ohjelmointirajapintoja, jotka tunnetaan myös grafiikkarajapintoina, kuten OpenGL ja Direct3D, jotka mahdollistavat 3D-grafiikan piirtämisen riippumatta siitä, mikä näytönohjain on käytössä [1] [2].

Nykypäivän pelimoottorit vaativat aina suurempia ja suurempia resursseja näytönohjaimilta ja pelinkehittäjät havaitsivat, että vanhat ohjelmointirajapinnat tekevät niiden resurssien hyödyntämisestä hankalaa. Tämä johti uusien, niin sanottujen modernien grafiikkarajapintojen kehittämiseen, joihin myös Vulkan kuuluu. Modernit grafiikkarajapinnat mahdollistavat paremmin tietokoneen kaikkien resurssien hyödyntämisen 3D-grafiikka piirtäessä [3].

Tämän opinnäytetyön tavoitteena on selvittää, mitä moderni grafiikkarajapinta tarkoittaa ja tutustua Vulkanin perusteisiin 3D-grafiikan piirtämisessä. Henkilökohtaisena tavoitteena on oppia Vulkanin käyttäminen.

2 GRAFIIKKARAJAPINTA

Grafiikkarajapinta on näytönohjaimen laiteajurista löytyvä ohjelmointirajapinta, joka tarjoaa abstraktion näytönohjaimen 3D-grafiikan piirtämiseen tarkoitettuihin ominaisuuksiin. Grafiikkarajapintoja käytetään erityisesti pelimoottoreissa 3D-grafiikan reaaliaikaiseen tuottamiseen.

2.1 Historia

Grafiikkarajapintoja on historian varrella ollut useita, ja niistä ensimmäiset olivat valmistaja- tai laiteriippuvaisia. 3D-grafiikan yleistyessä myös laiteriippumattomat rajapinnat, joilla voidaan tehdä ohjelmia, jotka toimivat useilla eri alustoilla, yleistyivät.

2.1.1 OpenGL

OpenGL on alun perin Silicon Graphics -nimisen yrityksen kehittämä avoin ja alustariippumaton grafiikkarajapinta, jonka ensimmäinen versio julkaistiin vuonna 1992 [1]. Nykyisin OpenGL kehitystä hallinnoi Khronos Group -konsortio, jonka tehtävänä on kehittää avoimia ohjelmointirajapintoja [4].

2.1.2 Direct3D

Direct3D on Microsoftin OpenGL rajapinnalle kehittämä kilpailija, jonka ensimmäinen versio julkaistiin vuonna 1995 ja on nykyisin yksi käytetyimpiä grafiikkarajapintoja pelialalla [2]. Direct3D:tä on mahdollista käyttää vain Microsoftin omilla alustoilla kuten Windowsilla. Direct3D:stä julkaistiin vuonna 2015 versio 12, joka kilpailee Vulkanin kanssa ns. modernin grafiikkarajapinnan asemasta [5].

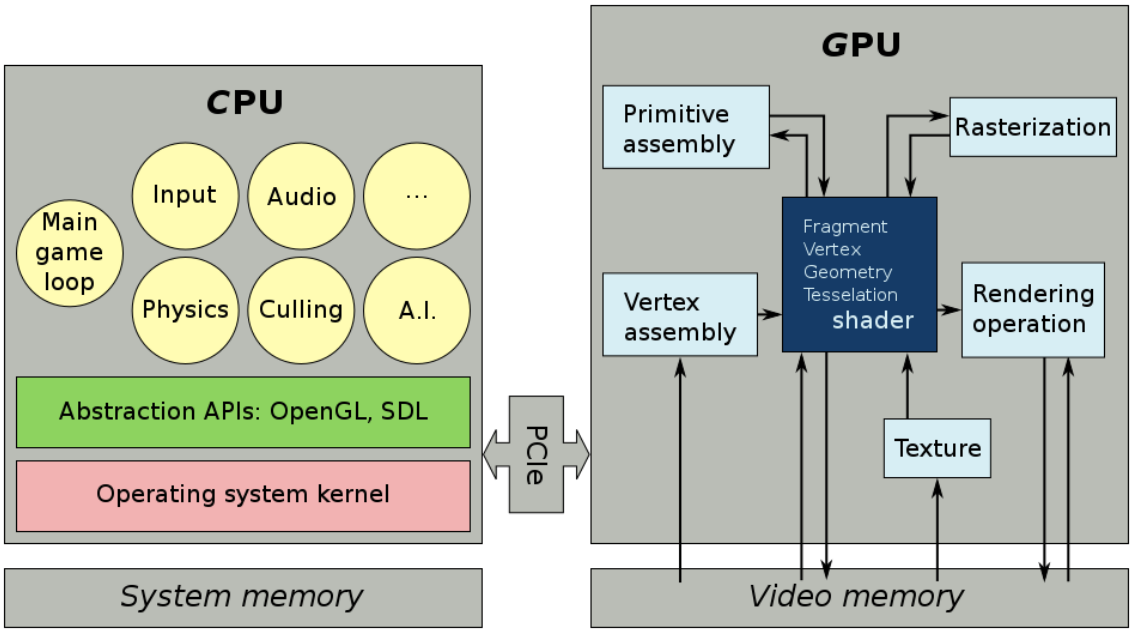
2.1.3 Mantle

Mantle oli AMD näytönohjainvalmistajan ja DICE-pelistudion yhteinen yritys tehdä uusi grafiikkarajapinta, jonka tarkoituksena oli helpottaa nykyisten näytönohjaimien ominaisuuksien hyödyntämistä ohjelmoijille [6]. Mantlesta ei koskaan julkaistu julkista versiota ja sen kehitys lakkautettiin, kun AMD lahjoitti sen Khronos Groupille käytettäväksi Vulkanin pohjana [7].

2.2 Näytönohjain

Grafiikkarajapinnat on tarkoitettu hallitsemaan tietokoneen näytönohjainta. Näytönohjain on jokaisesta tietokoneesta löytyvä komponentti, jonka tehtävä on piirtää kuva näytölle, ja se myös sisältää kaikki 3D-grafiikan piirtämiseen vaaditut ominaisuudet.

Näytönohjaimen rooli pelikäytössä on yleensä rajattu vain näytönohjaimelle tarkoitettujen ohjelmien eli varjostimien ajamiseen (käsitellään tarkemmin kohdassa 4.4.1), jotka määrittävät kuinka näytönohjain piirtää kuvan ruudulle (kuva 1). Kaikki muu pelin laskenta tapahtuu siis prosessorilla.



Kuva 1. Näytönohjaimen rooli peleissä.

2.3 Yleiset ominaisuudet

Kaikki grafiikkarajapinnat sisältävät ominaisuuden siirtää dataa näytönohjaimelle, kuten 3D-malleja ja tekstuureita. Tämän lisäksi niillä voi määrittää kuinka dataa käytetään esimerkiksi varjostimen kautta ja ne mahdollistavat lopullisen kuvan näyttämisen näytöllä. Kuitenkin näiden ominaisuuksien toteutus ja mahdollisten grafiikkarajapintaa tukevien alustojen määrä erottaa ne toisistaan.

2.4 Laiteläheisyys

Modernit grafiikkarajapinnat, kuten Direct3D 12 ja Vulkan, mainostavat olevansa laiteläheisiä, mikä tarkoittaa, että enemmän työtä siirretään näytönohjaimen laiteajurista ohjelman vastuulle, mikä mahdollistaa erityisesti paremman optimoinnin ja tarkemman hallinnan näytönohjaimen resursseista, mutta tästä johtuen myös monimutkaistaa kuvien piirtämiseen tarvittavaa koodia [5].

3 VULKAN

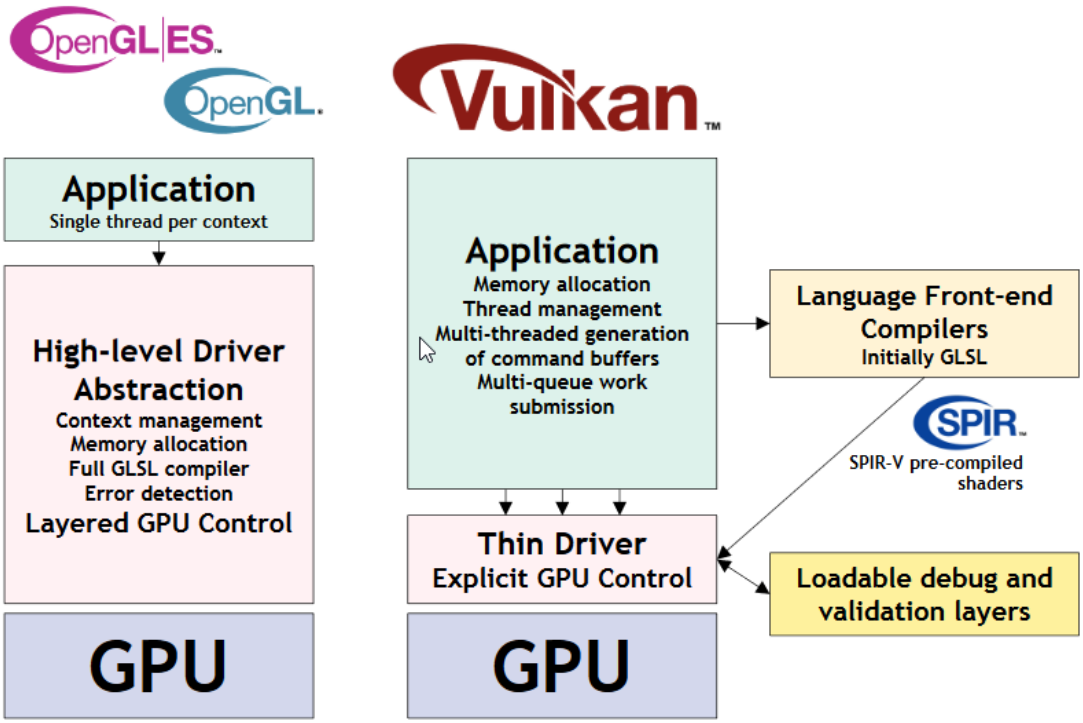
Vulkan on niin sanottu moderni laiteläheinen grafiikkarajapinta, jonka tarkoituksen on olla alustariippumaton ja helpottaa nykyisille näytönohjaimille optimoidun koodin tuottamista. Vulkanin tehtävänä on mahdollistaa 3D grafiikan piirtäminen näytönohjainta hyödyntämällä useilla eri käyttöjärjestelmillä ja laitteilla [8].

3.1 Historia

Vulkanin kehitys julkistettiin SIGGRAPH-tapahtumassa vuonna 2014, ja silloin se vielä tunnettiin nimellä glNext [9]. Vuonna 2015 Game Developer Conference tapahtumassa Vulkan julkistettiin virallisesti ja se sai nykyisen nimensä [10]. Vulkanin ensimmäinen julkinen versio julkaistiin 16. helmikuuta 2016. Se sisälsi version 1.0-määritelmän ja ohjelmistonkehityskirjaston [11]. Vulkan pohjautuu suurilta osin AMD:n lahjoittamaan Mantle-grafiikkarajapintaan, jota AMD oli itse kehittänyt käytettäväksi yrityksen omilla näytönohjaimilla [7].

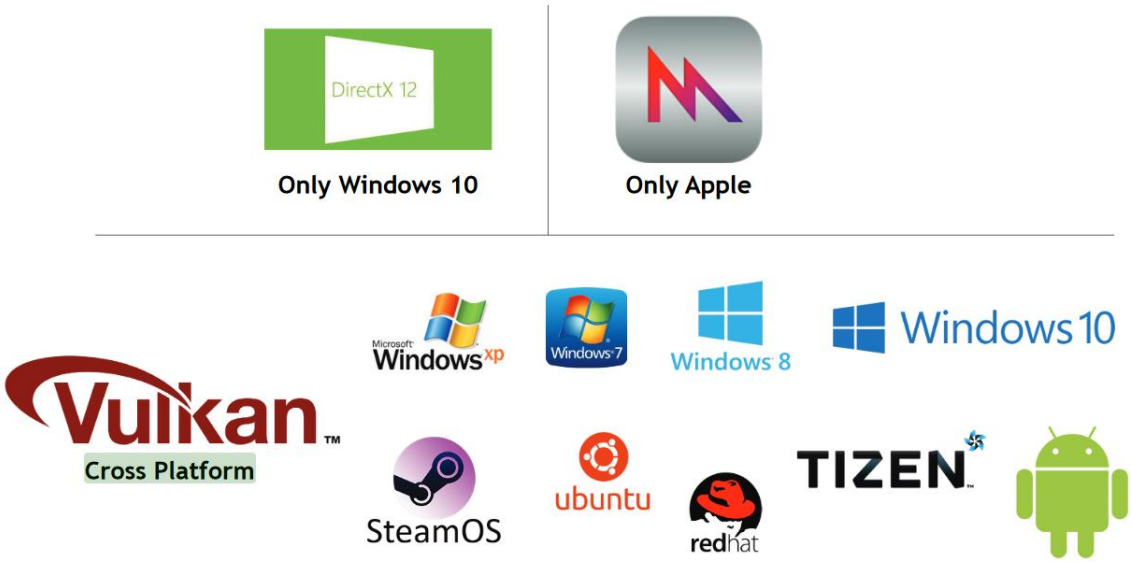
3.2 Erot muihin grafiikkarajapintoihin

Vulkan eroaa vanhemmista grafiikkarajapinnoista erityisesti sillä, että se on erittäin laiteläheinen ja piilottaa hyvin vähän nykyisen näytönohjaimen toimintaa ohjelmoijalta, kun taas vanhemmat grafiikkarajapinnat piilottavat enemmän näytönohjaimen toiminnasta laiteajureihin, mikä vaikeuttaa optimaalisen koodin tuottamista ja mahdollisten ohjelmistovirheiden löytämistä (kuva 2). Tästä johtuen Vulkanin käyttäminen vaatii huomattavasti enemmän koodin tuottamista ja ohjelmiston täytyy hyvin tarkasti määritellä, mitä näytönohjaimen ominaisuuksia se tulee hyödyntämään.



Kuva 2. Vulkan verrattuna vanhempiin grafiikkarajapintoihin [3].

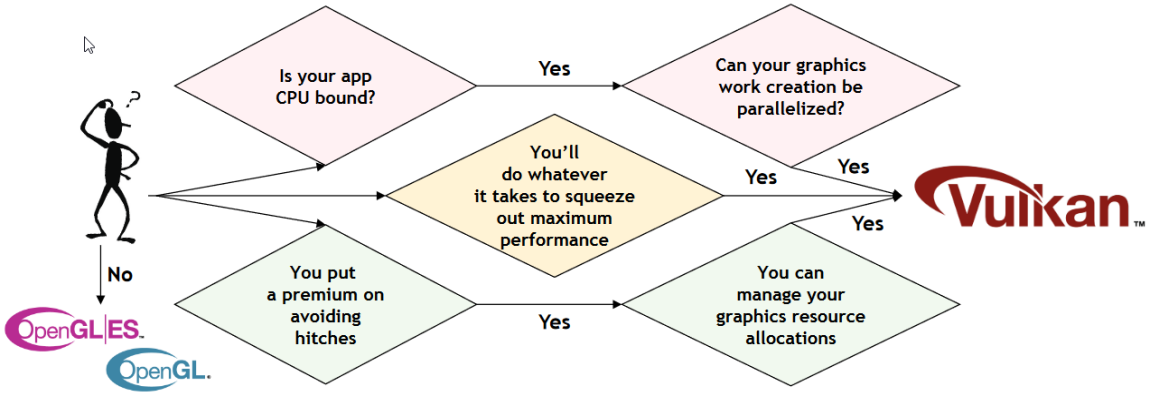
Vulkanin suurin ero muihin moderneihin grafiikkarajapintoihin on sen alustariippumattomuus, ja tästä syystä se on saatavilla useilla eri alustoilla (kuva 3).



Kuva 3. Vulkan verrattuna muihin moderneihin grafiikkarajapintoihin [3].

3.3 Vulkanin sopivuus

Vulkan ei välttämättä ole paras valinta kaikkiin sovelluksiin erityisesti vaikeamman käytön takia. Vulkania tulisi erityisesti käyttää, jos suorituskyky on tärkeämpää kuin koodin monimutkaisuus (kuva 4).



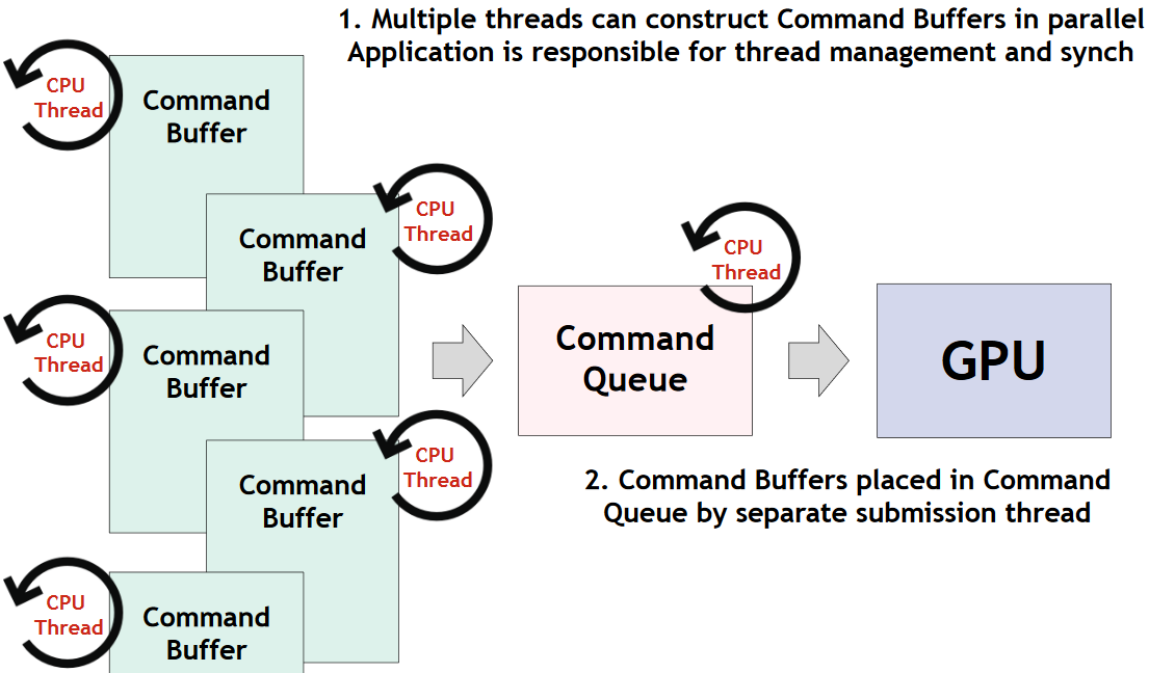
Kuva 4. Onko Vulkan sinulle sopiva [3]?

Vulkan on myös hyvä valinta siinä tilanteessa, että haluaa tukea mahdollisimman monta eri alustaa, ilman että tarvitsee käyttää useaa eri grafiikkarajapintaa (kuva 3).

3.4 Monisäikeistys

Monisäikeistyksellä tarkoitetaan työtaakan ajamista samanaikaisesti usealla eri prosessorin ytimellä, mikä mahdollistaa huomattavasti nopeamman prosessoinnin.

Vulkanissa grafiikan piirtämisen monisäikeistys on otettu huomioon tarjoamalla ohjelmoijalle mahdollisuus hyödyntää nykyprosessorien useita ytimiä (kuva 5).



Kuva 5. Monisäikeistys Vulkanissa [3].

Vulkanissa monisäikeistys tapahtuu antamalla komentoja komentopuskureihin, joita jokainen prosessorin ydin voi luoda itsenäisesti, ja tämän jälkeen keräämällä ne yhteiseen komentojonoon, minkä kautta ne siirtyvät näytönohjaimelle (kuva 5).

4 VULKANIN KÄYTTÄMINEN

Vulkanin ytimenä ovat oliot, joihin kaikki tarvittava tieto grafiikan piirtämistä varten tallennetaan. Muissa grafiikkarajapinnoissa osa tästä tiedosta on piilotettu laiteajureiden sisälle ja niiden sisältöön on todella vaikea vaikuttaa. Tämä tekee Vulkanin optimoinnista helpompaa, koska kaikki tarvittava tieto on täysin ohjelman hallinnassa.

Vulkanin olioista tärkeimmät ovat ns. instanssi- ja laiteolio, joista instanssiolio sisältää kaiken tiedon Vulkanin nykyisestä tilasta ja laiteolio sisältää kaiken tarvittavan grafiikan piirtämistä varten.

Vulkanissa muistinhallinta on ohjelmoijan vastuulla ja kaikki luodut oliot täytyy tuhota manuaalisesti, kun ohjelma ei niitä enää tarvitse.

Seuraavaksi tutustutaan syvällisemmin Vulkanin eri osa-alueisiin käyttämällä hyödyksi C++-koodiesimerkkejä. Koodiesimerkeissä hyödynnetään GLFW-kirjastoa, joka helpottaa ikkunoiden luomista ja Vulkanin alustamista alustariippumattomasti [12].

Vulkanilla ohjelmien kehitys vaatii myös ohjelmistokehityskirjaston, joka on saatavilla esimerkiksi LunarG-yrityksen tarjoamana ilmaiseksi [13].

4.1 Instanssi

Vulkanin instanssiolio on toinen kahdesta oliosta, joiden avulla lähes kaikkia Vulkanin funktioita käytetään. Instanssiolio on linkki ohjelman ja Vulkan-kirjaston välillä [14].

4.1.1 Instanssin luominen

Instanssiolion luomiseksi täytyy ensin kertoa Vulkan-kirjastolle hieman tietoa ohjelmasta. Aloitetaan luomalla `VkApplicationInfo`-olio, joka kertoo näytönohjaimen ajurille tietoa ohjelmasta [15].

```
VkApplicationInfo app = {};
app.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
app.pApplicationName = "Ohjelman nimi";
app.applicationVersion = 0;
app.pEngineName = "Moottorin nimi";
app.engineVersion = 0;
app.apiVersion = VK_API_VERSION_1_0;
```

Tässä alustetaan ensin tyhjä `VkApplicationInfo`-olio ja sen jälkeen asetetaan tarvittavat tiedot siihen. Tämä on myös esimerkki Vulkan-kirjaston C-kielen juurista, mikä pakottaa, että jokaiselle oliolle täytyy asettaa `sType`-arvo mikä kertoo olion tyypin. Tämän arvon asettamatta jättäminen tai väärin asettaminen aiheuttaa virheen ja mahdollisesti estää ohjelman toimimisen. Kuitenkin kaikki tyyppeihin liittyvät makrot seuraavat samaa kaavaa ja ovat näin ollen helposti pääteltävissä.

Seuraavaksi määritetään loput vaadittavat tiedot instanssiolion luomista varten [14].

```
VkInstance instance;

uint32_t extensionCount;
auto enabledExtensions =
glfwGetRequiredInstanceExtensions(&extensionCount);

VkInstanceCreateInfo instanceInfo = {};
instanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
instanceInfo.pApplicationInfo = &app;
instanceInfo.enabledExtensionCount = extensionCount;
instanceInfo.ppEnabledExtensionNames = enabledExtensions;
VkResult result = vkCreateInstance(&instanceInfo, nullptr, &instance);
```

Tässä määritetään `VkInstanceCreateInfo`-olio, johon asetetaan aikaisemmin määritetty olio ohjelman tiedosta [15]. Tämän lisäksi olioon täytyy määrittää, mitä laajennuksia sen tulisi käyttää. Vulkan-ohjelmalle täytyy aina määrittää alustariippuvaiset laajennukset, jotka mahdollistavat kommunikoinnin alustan ikkunoiden

kanssa. Tämän helpottamiseksi GLFW-kirjasto tarjoaa funktion, joka palauttaa tarvittavat laajennukset eri alustoilla [16]. Lopuksi `vkCreateInstance`-funktio luo instanssiolion määritetyillä tiedoilla ja asettaa sen `instance`-muuttujaan, joka tulisi säilyttää koko ohjelman elinajan [15]. Tämä funktio ja monet muut Vulkanin funktiot myös palauttavat palautusarvona virhekoodin, jossa `VK_SUCCESS`-arvo tarkoittaa onnistumista ja kaikki muut arvot jotakin virhettä [14].

Instanssi tulee tuhota manuaalisesti käyttämällä funktiota `vkDestroyInstance` [14].

4.1.2 Validointi ja virheenkorjaus

Vulkanissa virheentarkastus täytyy ottaa käyttöön manuaalisesti. Tämä on tehty siksi, että tarkastus ei ole ilmaista, vaan se hidastaa ohjelmaa.

Virheentarkastus otetaan käyttöön lisäämällä `vkCreateInstanceInfo`-olioon halutut virheentarkastusominaisuudet [14].

```
const std::vector<const char*> validationLayers = {
    "VK_LAYER_LUNARG_standard_validation",
};

instace_info.enabledLayerCount = validationLayers.size();
instace_info.ppEnabledLayerNames = validationLayers.data();
```

Tässä otetaan käyttöön niin kutsuttu standardivalidointikerros, joka ottaa käyttöön yleisimmät virheentarkastusominaisuudet. Myös muita virheentarkastusominaisuuksia on olemassa ja niitä voi ottaa käyttöön tarpeen mukaan [17].

Tuetut virheentarkastusominaisuudet voi listata funktiolla `vkEnumerateInstanceLayerProperties`, joka palauttaa listan kaikista kyseisellä laitteella/alustalla tuetuista ominaisuuksista. Oikeissa sovelluksissa tulisikin tarkistaa, ovatko vaaditut ominaisuudet tuettuja ennen niiden käyttämistä [15].

Tämän lisäksi `vkCreateInstanceInfo`-olioon listattuihin laajennuksiin täytyy lisätä `"VK_EXT_debug_report"`, joka mahdollistaa sen, että voidaan lukea Vulkanin palauttamat virheviestit [14]. Seuraavaksi rekisteröidään funktio, jolla luetaan palautetut virheviestit.

```
auto CreateDebugReportCallbackExt =
    reinterpret_cast<PFN_vkCreateDebugReportCallbackEXT>(
    vkGetInstanceProcAddr(instance, "vkCreateDebugReportCallbackEXT"));
VkDebugReportCallbackEXT debugReportCallback;

VkDebugReportCallbackCreateInfoEXT callbackCreateInfo = {};
callbackCreateInfo.sType =
VK_STRUCTURE_TYPE_DEBUG_REPORT_CALLBACK_CREATE_INFO_EXT;
callbackCreateInfo.pfnCallback = DebugCallback;
callbackCreateInfo.flags = VK_DEBUG_REPORT_ERROR_BIT_EXT

CreateDebugReportCallbackExt(instance, &callbackCreateInfo, nullptr,
&debugReportCallback);
```

Tässä rekisteröidään funktio, joka lukee mahdolliset virheviestit. Koska virheidentarkastus on laajennus, täytyy `vkCreateDebugReportCallbackEXT`-funktio ladata manuaalisesti käyttämällä instanssilaajennusfunktioiden lataamiseen tarkoitettua `vkGetInstanceProcAddr`-funktiota [14].

Tämän jälkeen luodaan `VkDebugReportCallbackCreateInfoEXT`-olio ja siihen asetetaan funktio, jota Vulkan kutsuu virheen sattuessa, ja vaaditut virheviestien tasot. Käyttämällä kyseistä oliota lopuksi rekisteröidään virheentarkastusfunktio käyttämällä aikaisemmin ladattua laajennusfunktiota ja tallennetaan `VkDebugReportCallbackEXT`-kahva mahdollista myöhempää käyttöä varten, kuten esimerkiksi rekisteröinnin poistaminen [14].

Virheviestien rekisteröinti tulee poistaa manuaalisesti käyttämällä `vkDestroyDebugReportCallbackEXT`-funktiota, joka kuten virheviestin lukemisen luontiin käytetty funktio tulee ladata manuaalisesti [14].

4.2 Laiteobjekti

Vulkanissa näytönohjaimen käyttäminen vaatii laiteobjektin luomisen, joka määrittää näytönohjaimelta vaaditut ominaisuudet ja mahdollistaa erilaisten komentojen (esim. kuvien piirtäminen) lähettämisen näytönohjaimelle [18, s. 3].

4.2.1 Fyysisen laitteen valinta

Laiteobjektin luomisessa täytyy ensimmäiseksi valita, mitä fyysistä näytönohjainta käytetään ja tallentaa se kahvaan [18, s. 7].

```
uint32_t gpuCount;
vkEnumeratePhysicalDevices(instance, &gpuCount, nullptr);

std::vector<VkPhysicalDevice> physicalDevices(gpuCount);
vkEnumeratePhysicalDevices(instance, &gpuCount,
physicalDevices.data());

VkPhysicalDevice gpu = physicalDevices[0];
```

Tässä listataan kaikki Vulkania tukevat näytönohjaimet ja tallennetaan listasta ensimmäisen kahva tulevaa käyttöä varten. Todellisuudessa, jos listassa on enemmän kuin yksi näytönohjain, olisi hyvä idea tarkistaa listasta paras näytönohjain käyttämällä `vkGetPhysicalDeviceProperties`- ja `vkGetPhysicalDeviceFeatures`-funktiota ja verrata näytönohjaimien ominaisuuksia ohjelmassa tarvittuihin ominaisuuksiin [15].

4.2.2 Jonoperheet

Vulkanissa lähes kaikki operaatiot aina kuvien piirtämisestä tekstuurien lataamiseen näytönohjaimen muistiin vaativat komentojonojen käyttämistä. Komentojonot on määritetty erilaisiin perheisiin, jotka tukevat vain tiettyjä operaatioita. Grafiikan piirtämiseksi täytyy löytää jonoperhe, joka tukee grafiikkaoperaatioita [18, s. 96–97.]

```
uint32_t queueCount;
vkGetPhysicalDeviceQueueFamilyProperties(gpu, &queueCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueCount);
vkGetPhysicalDeviceQueueFamilyProperties(gpu, &queueCount,
queueFamilies.data());

uint32_t graphicsQueue = 0;
for (size_t i = 0; i < queueFamilies.size(); i++)
{
    const auto& queueFamily = queueFamilies[i];
```

```
    if (queueFamily.queueCount >= 0 &&
        queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT)
    {
        graphicsQueue = i;
        break;
    }
}
uint32_t graphicsQueueIndex = graphicsQueue;
```

Tässä haetaan kaikki jonoperheet käyttämällä `vkGetPhysicalDeviceQueueFamilyProperties`-funktiota ja tarkistetaan, mikä niistä tukee grafiikkaoperaatioita vertaamalla sitä `VK_QUEUE_GRAPHICS_BIT`-arvoon [15]. Lopuksi sopiva jonoperhe tallennetaan muistiin myöhempää käyttöä varten [14].

Tämän lisäksi tulisi myös löytää jonoperhe, joka tukee kuvien esittämistä näytöllä, siinä tapauksessa, että kuvien piirtäminen näytölle on tavoitteena. Tämä onnistuu käyttämällä funktiota `vkGetPhysicalDeviceSurfaceSupportKHR` [14]. Tulee kuitenkin huomioida, että kyseisen funktion käyttäminen vaatii ikkunan pinnan luomisen, joka käydään läpi myöhemmin.

4.2.3 Laiteobjektin luominen

Laiteobjektin luomiseksi täytyy ensimmäiseksi määrittää kaikki vaaditut ominaisuudet. Ensimmäiseksi tulee määrittää halutut komentojonoperheet.

```
VkDeviceQueueCreateInfo queueCreateInfo = {};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = graphicsQueueIndex;
queueCreateInfo.queueCount = 1;
float queuePriorities = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriorities;
```

Tässä määritetään grafiikkaoperaatioita tukeva jonoperhe `VkDeviceQueueCreateInfo`-olioon ja annetaan sille prioriteetti, jossa 0,0 on alhaisin prioriteetti ja 1,0 korkein [15]. Tämän lisäksi olisi hyvä myös määrittää jonoperhe kuvien esittämiseksi niitä tilanteita varten, jolloin ei löydy jonoperhettä, joka tukee kuvien esittämistä ja grafiikkaoperaatioita yhtä aikaa.

Tämän jälkeen voidaan määrittää mahdolliset näytönohjaimelta halutut ominaisuudet `VkPhysicalDeviceFeatures`-olioon [15]. Lopuksi määritetään itse laiteobjektin luomiseen vaadittu olio ja luodaan laiteobjekti.

```
VkDeviceCreateInfo deviceCreateInfo = {};
deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
deviceCreateInfo.queueCreateInfoCount = 1;
deviceCreateInfo.pQueueCreateInfos = &queueCreateInfo;
deviceCreateInfo.enabledLayerCount = enabledLayerNames.size();
deviceCreateInfo.ppEnabledLayerNames = enabledLayerNames.data();
deviceCreateInfo.pEnabledFeatures = nullptr;

VkDevice device;
vkCreateDevice(gpu, &deviceCreateInfo, nullptr, &device);
```

Tässä määritetään `VkDeviceCreateInfo`-olio, johon asetetaan jonoperheen luomiseen vaadittu olio sekä mahdolliset virheentarkastuskerrokset [15]. Tämän lisäksi luodaan laiteobjeki käyttämällä `vkCreateDevice`-funktiota ja se tallennetaan tulevaa käyttöä varten [14].

Laiteobjekti tulee tuhota manuaalisesti käyttämällä `vkDestroyDevice`-funktiota [14].

4.2.4 Komentojonokahva

Laiteobjektin luominen luo myös automaattisesti kaikki pyydetyt komentojonoperheet ja niiden käyttämiseksi täytyy pyytää Vulkanilta kahva käyttämällä `vkGetDeviceQueue`-funktiota ja tallentamalla kahva myöhempää käyttöä varten [15].

4.3 Kuvan esittäminen

Ennen kuvien piirtämistä täytyy olla mahdollisuus esittää kuva näytöllä. Tätä varten Vulkan täytyy yhdistää ohjelman ikkunaan, jotta kuvat voidaan esittää ikkunassa. Tähän liittyen tulee myös luoda vaihtoketju joka sisältää kuvat, joihin lopullinen piirtäminen tapahtuu ja ne esitetään ikkunassa.

4.3.1 Ikkunan pinta

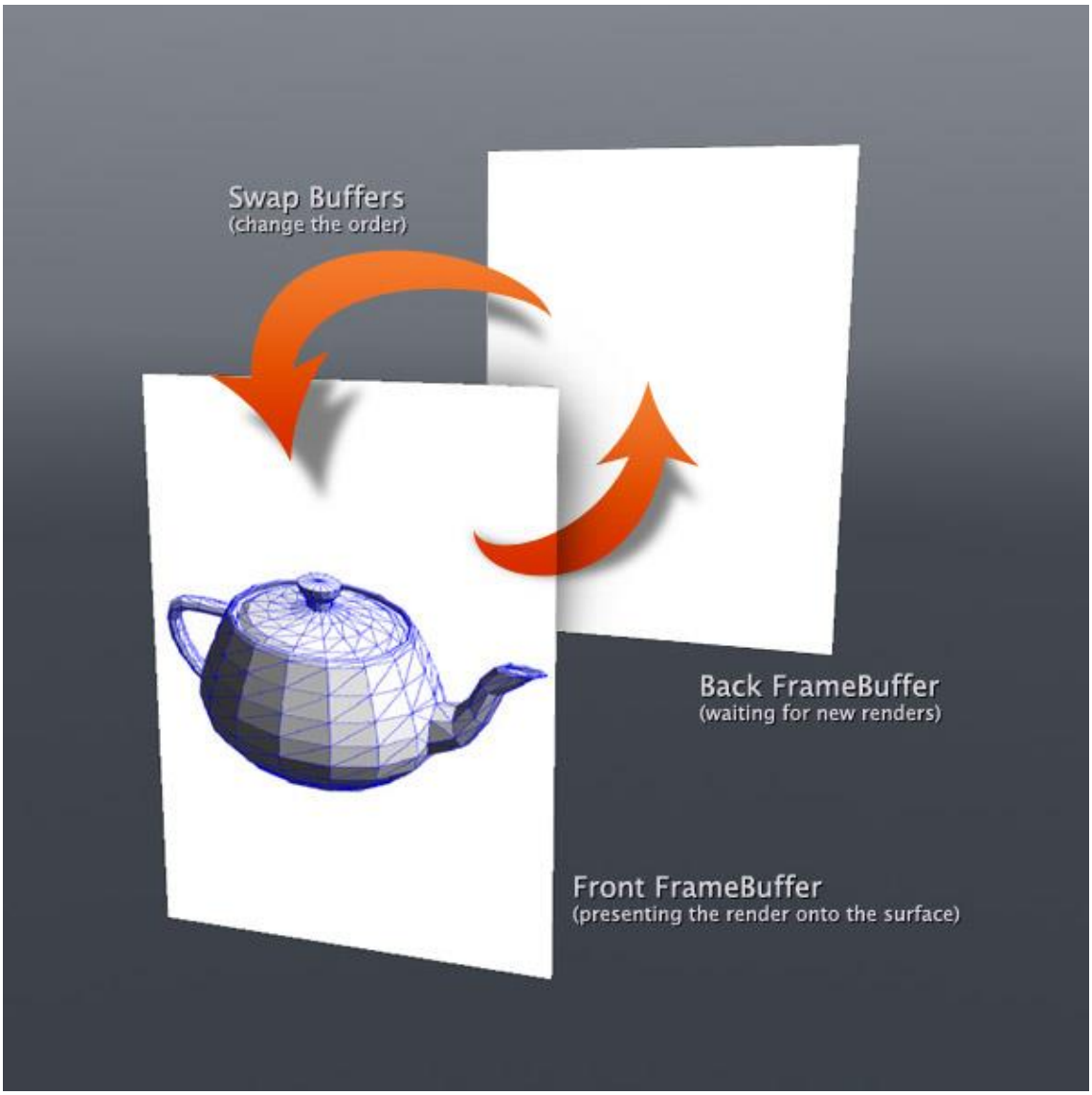
Vulkanissa ikkunassa olevaa aluetta, johon piirtäminen on mahdollista, kutsutaan pinnaksi [18, s. 138–139]. Ikkunan pinnan luominen on alustasta riippuvainen operaatio, mutta myös ikkunan luomiseen käytetty GLFW-kirjasto tarjoaa `glfwCreateWindowSurface`-funktion, joka luo ikkunan pinnan alustariippumattomasti [16].

Ikkunan pinta on myös mahdollista luoda manuaalisesti esimerkiksi Windows-alustalla käyttämällä `vkCreateWin32SurfaceKHR`-funktiota ja syöttämällä sille kaikki ikkunasta vaaditut tiedot [14].

Riippumatta tavasta, miten ikkunan pinta on luotu, se tulee manuaalisesti tuhota käyttämällä `vkDestroySurfaceKHR`-funktiota [14].

4.3.2 Vaihtoketju

Vaihtoketjulla tarkoitetaan tapaa esittää yleisesti kaksi tai joskus enemmän kuvaa niin, että yksi kuva on esillä näytöllä ja samanaikaisesti toiseen kuvaan piirretään ja tämän jälkeen kuvat vaihdetaan (kuva 6). Tämä estää käyttäjään näkemästä kuvaa kesken piirto-operaation, jolloin kuva voisi olla keskeneräinen [18, s. 143.]



Kuva 6. Esimerkki vaihtoketjun toiminnasta [19].

Vaihtoketjun luomiseksi täytyy ensimmäiseksi ottaa laiteobjektia luodessa käyttöön `"VK_KHR_swapchain"`-laajennus, joka mahdollistaa vaihtoketjun luomisen. Tämä siitä syystä, että kaikki näytönohjaimet eivät välttämättä tue kuvien esittämistä näytöllä, jos ne ovat esimerkiksi tarkoitettu serverikäyttöön [14]. Mahdollisuuden vaihtoketjun luomiseen voi tarkistaa `vkEnumerateDeviceExtensionProperties`-funktiolla [15].

Seuraavaksi tulee tarkistaa, mitä kuvaformaatteja näytönohjain tukee. Tämä onnistuu funktioilla `vkGetPhysicalDeviceSurfaceFormatsKHR` ja `vkGetPhysicalDeviceSurfacePresentModesKHR` [14]. Kun sopiva kuvaformaatti on löydetty ja tallennettu mahdollista tulevaa käyttöä varten, voidaan siirtyä luomaan vaihtoketju.

```
VkSurfaceCapabilitiesKHR surfaceCapabilities;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(gpu, surface,
&surfaceCapabilities);

VkExtent2D swapChainExtent;
if (surfaceCapabilities.currentExtent.width ==
std::numeric_limits<uint32_t>::max())
{
    swapChainExtent.width = windowWidth;
    swapChainExtent.height = windowHeight;
}
else
{
    swapChainExtent = surfaceCapabilities.currentExtent;
}

uint32_t desiredNumberOfSwapChainImages =
surfaceCapabilities.minImageCount + 1;
if (surfaceCapabilities.maxImageCount > 0 &&
desiredNumberOfSwapChainImages > surfaceCapabilities.maxImageCount)
{
    desiredNumberOfSwapChainImages = surfaceCapabilities.maxImageCount;
}

VkSwapchainCreateInfoKHR swapChainCreateInfo = {};
swapChainCreateInfo.sType =
VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
swapChainCreateInfo.surface = surface;
swapChainCreateInfo.minImageCount = desiredNumberOfSwapChainImages;
swapChainCreateInfo.imageFormat = VK_FORMAT_B8G8R8A8_UNORM;
swapChainCreateInfo.imageColorSpace =
VK_COLOR_SPACE_SRGB_NONLINEAR_KHR;
swapChainCreateInfo.imageExtent = swapChainExtent;
swapChainCreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
swapChainCreateInfo.preTransform =
surfaceCapabilities.currentTransform;
swapChainCreateInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
swapChainCreateInfo.imageArrayLayers = 1;
swapChainCreateInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
swapChainCreateInfo.queueFamilyIndexCount = 0;
swapChainCreateInfo.pQueueFamilyIndices = nullptr;
swapChainCreateInfo.presentMode = VK_PRESENT_MODE_FIFO_KHR;
swapChainCreateInfo.oldSwapchain = VK_NULL_HANDLE;
swapChainCreateInfo.clipped = VK_TRUE;

vkCreateSwapchainKHR(device, &swapChainCreateInfo, nullptr,
&swapChain);
```

Tässä kysytään aluksi näytönohjaimelta, mitä rajoituksia vaihtoketjun luomiseen liittyy käyttämällä `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`-funktiota ja käytetään tätä informaatiota selvittämään vaihtoketjun kuvien ihanteellinen resoluutio ja montako kuvaa tulisi luoda, jotta kuvien vaihtaminen onnistuu [14]. Sen jälkeen

luodaan `VkSwapchainCreateInfoKHR`-olio ja siihen asetetaan haluttujen kuvien määrä, resoluutio ja kuvaformaatti [14]. Tässä esimerkissä asetetaan arvot yleisimmin saatavilla oleviin kuvaformaatteihin ja asetuksiin, mutta todellisuudessa tulisi tarkistaa paras saatavilla oleva formaatti [18, s. 143–147]. Olioon tulee myös määrittää vanhentunut vaihtoketju, siinä tilanteessa, että vaihtoketjua ollaan luomassa uudelleen esimerkiksi ikkunan koon muutoksen takia. Lopuksi vaihtoketju luodaan käyttämällä `vkCreateSwapchainKHR`-funktiota [14].

Lopuksi haetaan vaihtoketjun luonnin yhteydessä luodut kuvat `vkGetSwapchainImagesKHR`-funktiolla [14].

Vaihtoketju tulee tuhota manuaalisesti käyttämällä `vkDestroySwapchainKHR`-funktiota [14].

4.3.3 Kuvanäkymä

Vulkanissa kaikkien kuvien muokkaamiseen tulee luoda kuvanäkymä, joka on kirjaimellisesti näkymä kyseiseen kuvaan. Kuvanäkymä määrittää, mitä aluetta kuvasta halutaan muokata ja miten kuvaa tulisi käsitellä (esim. 2-ulotteinen tekstuuri). Myöskin vaihtoketjun kuvien käyttämiseen tulee luoda niille kuvanäkymä.

```
for (size_t i = 0; i < swapChainImageCount; i++)
{
    VkImageViewCreateInfo colorImageView = {};
    colorImageView.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    colorImageView.format = VK_FORMAT_B8G8R8A8_UNORM;
    colorImageView.components.r = VK_COMPONENT_SWIZZLE_IDENTITY;
    colorImageView.components.g = VK_COMPONENT_SWIZZLE_IDENTITY;
    colorImageView.components.b = VK_COMPONENT_SWIZZLE_IDENTITY;
    colorImageView.components.a = VK_COMPONENT_SWIZZLE_IDENTITY;
    colorImageView.subresourceRange.aspectMask =
VK_IMAGE_ASPECT_COLOR_BIT;
    colorImageView.subresourceRange.baseMipLevel = 0;
    colorImageView.subresourceRange.levelCount = 1;
    colorImageView.subresourceRange.baseArrayLayer = 0;
    colorImageView.subresourceRange.layerCount = 1;
    colorImageView.viewType = VK_IMAGE_VIEW_TYPE_2D;
    colorImageView.image = swapChainImages[i];

    vkCreateImageView(device, &colorImageView, nullptr,
&swapChainBuffers[i].imageView);
```

```
}
```

Tässä luodaan yksinkertainen kuvanäkymä jokaiselle vaihtoketjun kuvalle kuvien piirtämistä varten. Aluksi luodaan `VkImageViewCreateInfo`-olio ja siihen asetetaan yksinkertaiset arvot, jotka mahdollistavat yksinkertaisen koko kuvaan piirtämisen [14]. Tämän jälkeen kuvanäkymä luodaan `vkCreateImageView`-funktiolla [15].

Kuvanäkymä tulee tuhota manuaalisesti käyttämällä `vkDestroyImageView`-funktiota [14].

4.4 Grafiikkaliukuhihna

Grafiikkaliukuhihnalla tarkoitetaan Vulkanissa niitä operaatioita, jotka läpikäymällä piirrettävä data, kuten tekstuurit ja 3D-mallit, muuttuvat pikseleiksi ruudulla. Tähän sisältyvät mm. varjostimet, piirtokierrokset ja kuvapuskurit. Toisin kuin muissa grafiikkarajapinnoissa, Vulkanin grafiikkaliukuhihna ei sisällä mitään oletusarvoja. Tästä johtuen grafiikkaliukuhihnaa luodessa täytyykin kaikki halutut ominaisuudet määrittää eksplisiittisesti [18, s. 226–229.]

4.4.1 Varjostimet

Varjostimella tarkoitetaan ohjelmaa, joka ajetaan näytönohjaimella ja sen tehtävänä on määrittää, kuinka näytönohjaimelle lähetetty data muutetaan pikseleiksi [20]. Erityisesti 3D-grafiikassa tähän yleensä myös sisältyy valaistuksen ja varjojen laskenta mistä varjostimen nimi myös juontuu.

Vulkanissa varjostimet yleensä kirjoitetaan GLSL-kieltä käyttäen ja ne tulee ennen lataamista kääntää SPIR-V-formaattiin [18, s. 165–173], joka onnistuu esimerkiksi Vulkanin ohjelmistokehityspaketin mukana tulevalla glslangValidator-sovelluksella [21]. Vulkan tukee useita erityyppisiä varjostimia, kuten esimerkiksi verteksi, pikseli ja geometria varjostimia. Näistä kuvan piirtämiseksi verteksi ja pikseli varjostimet ovat pakolliset.

Varjostimien lataaminen on suhteellisen yksinkertaista. Ensin varjostin luetaan tiedostosta, ja sen jälkeen kerrotaan Vulkanille, minkä tyyppinen varjostin on kyseessä.

```
std::ifstream file;
file.open("triangle.vert.spv", std::ios::binary | std::ios::ate);
uint32_t size = static_cast<uint32_t>(file.tellg());
file.seekg(0, std::ios::beg);

std::vector<char> vertexData;
vertexData.resize(size);
file.read(vertexData.data(), size);
file.close();

VkShaderModuleCreateInfo moduleCreateInfo = {};
moduleCreateInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
moduleCreateInfo.codeSize = size;
moduleCreateInfo.pCode =
reinterpret_cast<uint32_t*>(vertexData.data());

VkShaderModule vertexModule;
vkCreateShaderModule(vulkan.device, &moduleCreateInfo, nullptr,
&vertexModule);

VkPipelineShaderStageCreateInfo vertexShaderStageInfo = {};
vertexShaderStageInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
vertexShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
vertexShaderStageInfo.module = vertexModule;
vertexShaderStageInfo.pName = "main";
```

Tässä luetaan aluksi varjostimen koodi tiedostosta (liite 1). Tämän jälkeen luodaan `VkShaderModuleCreateInfo`-olio, jonka avulla varjostimen koodista luodaan `VkShaderModule`-kahva [15]. `VkPipelineShaderStageCreateInfo`-olioon määritetään varjostimen tyyppi, aiemmin luotu kahva ja funktion nimi, josta varjostimen koodin ajaminen aloitetaan [15]. Samalla periaatteella tulisi myös ladata pikselivarjostin (liite 2) ja tallentaa molemmat `VkPipelineShaderStageCreateInfo`-oliot myöhempää grafiikkaliukuhihnan luomista varten.

Grafiikkaliukuhihnan luonnin jälkeen tulee `VkShaderModule`-oliot tuhota käyttämällä `vkDestroyShaderModule`-funktiota [14].

4.4.2 Piirtokierrokset

Vulkanissa piirtokierroksilla tarkoitetaan tapaa piirtää kuva kerroksittain. Esimerkiksi ensimmäinen kierros voisi piirtää yksinkertaisen kuvan ja toinen kierros lisätä kuvaan valaistuksen. Piirtokierroksien määrä on täysin riippuivainen ohjelmasta, mutta jokaisessa ohjelmassa täytyy olla vähintään yksi kuvan piirtämistä varten [18, s. 230.]

Yksinkertainen piirtokierros luodaan aluksi määrittämällä siihen liittyvien kuvien tyypit ja tämän jälkeen määrittämällä jokainen piirtokierros yksitellen, jonka jälkeen nämä yhdistävä piirtokierros voidaan luoda. Monimutkaisemmassa piirtokierroksessa voitaisiin myös määrittää mahdolliset yksittäisen piirtokierroksien toisistaan riippuvuudet, jos esimerkiksi toinen piirtokierros on riippuvainen ensimmäisen lopputuloksesta [18, s. 401–423].

```
VkAttachmentDescription attachment = {};
attachments[0].format = format;
attachments[0].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
attachments[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[0].finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Tässä esimerkissä luodaan aluksi määritelmä yhdelle väridataa sisältävälle kuvalle käyttämällä `VkAttachmentDescription`-oliota [15]. Muun tyyppisiä kuvia olisivat mm. syvyysdataa sisältävä kuva. Olioon asetetaan aluksi vaihtoketjua luodessa valittu kuvaformaatti ja tämän jälkeen määritetään, mitä Vulkan tekee kuvalle piirron aloitusvaiheessa ja sen lopussa. Tässä tapauksessa `VK_ATTACHMENT_LOAD_OP_CLEAR`-arvo tarkoittaa, että kuva tyhjennetään piirron alkuvaiheessa ja `VK_ATTACHMENT_STORE_OP_STORE`-arvo kehottaa Vulkania tallentamaan kuvan muistiin piirron lopussa. Lopuksi määritetään, että kuvaa käytetään kuvien esittämisen lähteenä.

Seuraavaksi tulee määrittää kuvamääritelmälle viittaus, jotta yksittäiset piirtokierrokset voivat viitata siihen.

```
VkAttachmentReference colorReference = {};
colorReference.attachment = 0;
colorReference.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Tässä määritetään yksinkertainen viittaus käyttämällä `VkAttachmentReference`-olio, johon asetetaan kuvamääritelmän indeksi ja tyyppi [15]. Tässä tapauksessa kuvamääritelmiä on vain yksi, joten indeksi on nolla.

Seuraavaksi tulee määrittää yksittäinen piirtokierros.

```
VkSubpassDescription subpass = {};
subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount = 1;
subpass.pColorAttachments = &colorReference;
```

Yksittäinen piirtokierros määritetään käyttämällä `VkSubpassDescription`-oliota [15]. Tässä erimerkissä siihen asettaan vain yksi väridataa sisältävän kuvan viittaus ja määritetään, että se on grafiikan piirtämiseen tarkoitettu piirtokierros.

Lopuksi voidaan luoda itse piirtokierros.

```
VkRenderPassCreateInfo renderPassInfo = {};
renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
renderPassInfo.attachmentCount = 1;
renderPassInfo.pAttachments = attachment;
renderPassInfo.subpassCount = 1;
renderPassInfo.pSubpasses = &subpass;
renderPassInfo.dependencyCount = 0;
renderPassInfo.pDependencies = nullptr;

VkRenderPass renderPass;
vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass);
```

Tässä määritetään aluksi piirtokierros käyttämällä `VkRenderPassCreateInfo`-oliota [15]. Siihen asetetaan kuvamääritelmät ja kaikki yksittäiset piirtokierrokset, sekä mahdolliset piirtokierroksien riippuvuudet. Lopuksi piirtokierros luodaan `vkCreateRenderPass`-funktiolla ja se tallennetaan `VkRenderPass`-kahvaan [15].

Piirtokierrokset tulee manuaalisesti tuhota käyttämällä `vkDestroyRenderPass`-funktiota [14].

4.4.3 Kuvapuskurit

Kuvapuskuri on olio, jonka tarkoituksena on määrittää, mihin kuviin grafiikkaliukuhihna piirtää [18, s. 237]. Kuvapuskuriolio luodaan viittaamalla käytettävään piirtokierrokseen ja vaihtoketjun kuvan kuvanäkymään. Kuvapuskuria on mahdollista käyttää myös muiden piirtokierroksien kanssa, jos ne sisältävät samanlaiset viittaukset käytettyihin kuviin. On myös mahdollista määrittää kuvia, joita ei välttämättä käytetä siinä tapauksessa, että halutaan tukea useaa eri piirtokierrosta, joilla on eri määrä käytössä olevia kuvia [18, s. 238].

```
std::vector<VkFramebuffer> framebuffers(swapChainBuffers.size());

for (size_t i = 0; i < swapChainBuffers.size(); i++)
{
    VkImageView attachments[] = { swapChainImageViews[i] };
    VkFramebufferCreateInfo framebufferCreateInfo = {};
    framebufferCreateInfo.sType =
VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferCreateInfo.renderPass = renderPass;
    framebufferCreateInfo.attachmentCount = 1;
    framebufferCreateInfo.pAttachments = attachments;
    framebufferCreateInfo.width = windowWidth;
    framebufferCreateInfo.height = windowHeight;
    framebufferCreateInfo.layers = 1;

    vkCreateFramebuffer(device, &framebufferCreateInfo, nullptr,
&framebuffers[i]);
}
```

Tässä luodaan vaihtoketjun kuvien määrän verran kuvapuskureita. `VkFramebufferCreateInfo`-olioon [15] määritetään käytettävä piirtokierros ja vaihtoketjun kuvanäkymä. Koska kuvapuskuria käytetään kuvien piirtämiseen, määritetään siihen myös vaihtoketjun kuvien eli ikkunan resoluution. Lopuksi jokainen kuvapuskuri luodaan `vkCreateFramebuffer`-funktiolla [15].

Kuvapuskurit tulee manuaalisesti tuhota käyttämällä `vkDestroyFramebuffer`-funktiota [14].

4.4.4 Grafiikkaliukuhihnan tila

Kuten aiemmin mainittu, Vulkanin grafiikkaliukuhihna ei sisällä mitään oletusarvoja, joten kaikki arvot tulee määrittää itse. Grafiikkaliukuhihnan määritettävään tilaan kuuluu mm. Verteksidatan formaatti, piirtämisen tyyppi (esim. kolmio tai viiva), kuvaportin koko ja syvyystestauksen asetukset. Osa grafiikkaliukuhihnan tilasta on myös mahdollista määrittää ns. dynaamiseksi tilaksi, eli se täytyy määrittää aina ennen kuvan piirtämistä. Tämä myös tarkoittaa, että sitä voidaan muuttaa kesken ohjelman ajon ilman grafiikkaliukuhihnan uudelleen luomista.

Yksinkertaisen kuvan piirtämiseksi grafiikkaliukuhihnan tilan voi määrittää seuraavasti.

```
VkPipelineInputAssemblyStateCreateInfo inputAssemblyState = {};
inputAssemblyState.sType =
VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
inputAssemblyState.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
```

Tässä määritetään piirtämisen tyypiksi lista kolmiota käyttämällä `VkPipelineInputAssemblyStateCreateInfo`-oliota [15].

Seuravaksi määritetään rasterisoinnin ominaisuudet.

```
VkPipelineRasterizationStateCreateInfo rasterizationState = {};
rasterizationState.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
rasterizationState.polygonMode = VK_POLYGON_MODE_FILL;
rasterizationState.cullMode = VK_CULL_MODE_NONE;
rasterizationState.frontFace = VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizationState.depthClampEnable = VK_FALSE;
rasterizationState.rasterizerDiscardEnable = VK_FALSE;
rasterizationState.depthBiasEnable = VK_FALSE;
rasterizationState.lineWidth = 1.0f;
```

Tässä määritetään rasterisoinnin ominaisuudet käyttämällä `VkPipelineRasterizationStateCreateInfo`-oliota [15]. Aluksi määritettään miten kolmiot tulisi piirtää. Tässä esimerkissä `VK_POLYGON_MODE_FILL`-arvo tarkoittaa, että kolmiot piirretään täytettyinä. Muita vaihtoehtoja ovat pelkät reunaviivat tai pisteet. Tämän jälkeen määritetään tulisiko piilossa olevia kolmiota jättää piirtämättä ja myöskin määritetään minkälaiset kolmiot ovat mahdollisesti piilossa. Tässä esimerkissä

piiretään yksinkertaisesti kaikki kolmiot. Lopuksi määritetään monimutkaisemmat ominaisuudet pois päältä yksinkertaista piirtoa varten.

Seuraavaksi määritetään kuinka värit tulisi sekoittaa.

```
VkPipelineColorBlendAttachmentState blendAttachmentState[1] = {};
blendAttachmentState[0].colorWriteMask = 0xf;
blendAttachmentState[0].blendEnable = VK_FALSE;

VkPipelineColorBlendStateCreateInfo colorBlendState = {};
colorBlendState.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlendState.attachmentCount = 1;
colorBlendState.pAttachments = blendAttachmentState;
```

Tässä määritettään aluksi värien sekoittaminen `VkPipelineColorBlendAttachmentState`-olioon. Tässä esimerkissä ei käytetä värien sekoitusta, joten se laitetaan pois käytöstä. Tämän jälkeen värien sekoitus tulee määrittää grafiikkaliukuhihnalle `VkPipelineColorBlendStateCreateInfo`-olioon [15.]

Seuraavaksi määritetään kuvaportti ja siihen piirrettävä alue.

```
std::vector<VkDynamicState> dynamicStates = {
VK_DYNAMIC_STATE_VIEWPORT, VK_DYNAMIC_STATE_SCISSOR};

VkPipelineDynamicStateCreateInfo dynamicState = {};
dynamicState.sType =
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicState.pDynamicStates = dynamicStates.data();
dynamicState.dynamicStateCount = dynamicStates.size();

VkPipelineViewportStateCreateInfo viewportState = {};
viewportState.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
viewportState.viewportCount = 1;
viewportState.scissorCount = 1;
```

Tässä määritetään esimerkin vuoksi kuvaportti ja siihen piirrettävä alue käyttämään dynaamista tilaa `VkPipelineDynamicStateCreateInfo`-olion avulla [15]. Sen lisäksi määritetään grafiikkaliukuhihnassa käytettäväksi yksi kuvaportti käyttämällä `VkPipelineViewportStateCreateInfo`-oliota [15]. Tämä sen takia, että joillakin näytönohjaimilla on mahdollista käyttää useampaa kuvaporttia, mutta tämä vaatii laajennuksen ottamisen käyttöön laitteen luomisen yhteydessä. Lopullinen kuvaportti määritetään piirron yhteydessä, koska se käyttää dynaamista tilaa.

Seuraavaksi määritetään mahdolliset moniotannan ominaisuudet.

```
VkPipelineMultisampleStateCreateInfo multisampleState = {};
multisampleState.sType =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampleState.pSampleMask = nullptr;
multisampleState.rasterizationSamples = VK_SAMPLE_COUNT_1_BIT;
```

Tässä esimerkissä ei käytetä moniotantaa, joten se määritetään pois käytöstä käyttämällä `VkPipelineMultisampleStateCreateInfo`-oliota [15].

Seuraavaksi täytyy määrittää grafiikkaliukuhihnan asettelu, jonka avulla voidaan siirtää arvoja varjostimeen. Tähän tutustutaan tarkemmin myöhemmin, joten tässä luodaan vain tyhjä asettelu.

```
VkPipelineLayout pipelineLayout
VkPipelineLayoutCreateInfo pipelineLayoutCreateInfo = {};
pipelineLayoutCreateInfo.sType =
VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;

vkCreatePipelineLayout(vulkan.device, &pipelineLayoutCreateInfo,
nullptr, &pipelineLayout);
```

Tässä luodaan tyhjä grafiikkaliukuhihnan asettelu käyttämällä `VkPipeline-LayoutCreateInfo`-oliota ja tallennetaan se `VkPipelineLayout`-kahvaan [15]. Grafiikkaliukuhihnan asettelu tulee manuaalisesti tuhota käyttämällä `vkDestroyPipelineLayout`-funktiota [14].

Grafiikkaliukuhihna tarvitsee myös määritelmän sille syötettävästä verteksidatasta. Tämä käydään läpi myöhemmin datan siirtoon liittyvässä osiossa, joten tässä luodaan vain tyhjän määritelmä.

```
VkPipelineVertexInputStateCreateInfo vertexInputState = {};
vertexInputState.sType =
VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
```

Tässä luodaan tyhjä verteksidatan määritys käyttämällä `VkPipelineVertexInputStateCreateInfo`-oliota [15].

Tämän lisäksi voitaisiin myös määrittää esimerkiksi syvyyspuskurin ominaisuudet käyttämällä `VkPipelineDepthStencilStateCreateInfo`-oliota, mutta koska tämä esimerkki ei käytä syvyysdataa kuvan piirtämisessä, voidaan lopullisen grafiikkaliukuhihnan luonnissa jättää arvo asettamatta.

4.4.5 Grafiikkaliukuhihnan luominen

Grafiikkaliukuhihan luomiseksi otetaan kaikki edellä mainitut ominaisuudet ja yhdistetään ne lopullisen grafiikkahihnan luomiseksi.

```
VkGraphicsPipelineCreateInfo pipelineCreateInfo = {};
pipelineCreateInfo.sType =
VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineCreateInfo.layout = pipelineLayout;
pipelineCreateInfo.renderPass = renderPass;
pipelineCreateInfo.stageCount = shaderStages.size();
pipelineCreateInfo.pStages = shaderStages.data();
pipelineCreateInfo.pVertexInputState = &vertexInputState;
pipelineCreateInfo.pInputAssemblyState = &inputAssemblyState;
pipelineCreateInfo.pRasterizationState = &rasterizationState;
pipelineCreateInfo.pColorBlendState = &colorBlendState;
pipelineCreateInfo.pMultisampleState = &multisampleState;
pipelineCreateInfo.pViewportState = &viewportState;
pipelineCreateInfo.pDepthStencilState = nullptr;
pipelineCreateInfo.pDynamicState = &dynamicState;

VkPipeline pipeline;

vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1,
&pipelineCreateInfo, nullptr, &pipeline);
```

Tässä määritetään grafiikkaliukuhihna käyttämällä `VkGraphicsPipelineCreateInfo`-oliota [15]. Olioon asetetaan aluksi aikaisemmin luodut grafiikkaliukuhihnan asetelmamääritys ja piirtokierros. Tämän jälkeen määritetään aikaisemmin ladatut varjostimet ja lopuksi määritetään kaikki aikaisemmin määritelty grafiikkaliukuhihnan tila. Tämän jälkeen grafiikkaliukuhihna luodaan `vkCreateGraphicsPipelines`-funktiolla [15]. Tämä funktio eroaa jonkin verran muista Vulkanin funktiosta siinä, että sillä voidaan luoda useita grafiikkaliukuhihnoja yhdellä kertaa, mistä johtuen siihen tulee määrittää luotavien liukuhihnojen määrä (tässä tapauksessa yksi). Tässä myöskin voitaisiin käyttää grafiikkaliukuhihnan välimuistia (`VkPipelineCache`), joka mahdollistaa mm. grafiikkaliukuhihnojen tallentamisen tiedostoon ja tämän tiedoston hyödyntämistä myöhemmässä vaiheessa grafiikkaliukuhihnan luonnin nopeuttamiseksi [18, s. 181–186].

Grafiikkaliukuhihna tulee manuaalisesti tuhota käyttämällä `vkDestroyPipeline`-funktiota [14].

4.5 Komennot

Vulkanissa komentoja, kuten piirtäminen ja datan siirtäminen näytönohjaimelle, ei kutsuta suoraan, vaan ne kerätään komentopuskureihin, jotka komentojonojen kautta lähetetään näytönohjaimelle. Tämä mahdollistaa sen, että komennot ja niiden tarvitsema laskenta, voidaan tallentaa käyttämällä useita säikeitä yhtäaikaisesti ja tämän jälkeen lähettää yhtenä kokonaisuutena näytönohjaimelle [18, s. 97–105.]

4.5.1 Komentopooli

Vulkanissa komentopuskureita ei luoda itsessään, vaan ne luodaan komentopoolien kautta, joiden tehtävä on hallita komentopuskureita ja niiden vaatimaa muistia. Komentopoolit ovat linkitettyjä tiettyyn komentojonoperheeseen ja niistä luotuja komentopuskureita voikin lähettää vain saman jonoperheen komentojonoihin.

Ennen komentopuskureiden käyttöä tuleekin siis luoda komentopooli.

```
VkCommandPool commandPool;

VkCommandPoolCreateInfo commandPoolCreateInfo = {};
commandPoolCreateInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
commandPoolCreateInfo.queueFamilyIndex = graphicsQueueIndex;
vkCreateCommandPool(device, &commandPoolCreateInfo, nullptr,
&commandPool);
```

Tässä määritetään komentopooli käyttämällä `VkCommandPoolCreateInfo`-oliota ja annetaan sille aiemmin määritetty grafiikkakomentoja tukeva jonoperhe. Tämän jälkeen komentopooli luodaan käyttämällä funktiota `vkCreateCommandPool` [15.]

Komentopooli tulee manuaalisesti tuhota käyttämällä `vkDestroyCommandPool`-funktiota [14].

4.5.2 Komentopuskurit

Komentopuskurit ovat avain työn lähettämiseksi näytönohjaimelle, ja niiden avulla tapahtuu esimerkiksi kuvien piirtäminen. Komentopuskuri luodaan varaamalla se komentopoolista [18, s. 97–100].

```
VkCommandBufferAllocateInfo commandBufferAllocateInfo = {};
commandBufferAllocateInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
commandBufferAllocateInfo.commandPool = commandPool;
commandBufferAllocateInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
commandBufferAllocateInfo.commandBufferCount = 1;

vkAllocateCommandBuffers(device, &commandBufferAllocateInfo,
&commandBuffer);
```

Tässä määritetään luotavaksi yksi komentopuskuri käyttämällä `VkCommandBufferAllocateInfo`-oliota ja siihen asetetaan aiemmin luotu komentopooli [15].

Komentopuskurit lähetetään näytönohjaimelle käyttämällä `vkQueueSubmit`-funktiota, mutta tähän tutustutaan tarkemmin kuvan piirtämistä käsittelevässä osiossa [15].

Komentopuskureita ei tarvitse tuhota, koska komentopoolin tuhoaminen tuhoaa myös komentopuskurille varatun muistin. Komentopuskureita voidaan kuitenkin palauttaa komentopoolille käyttämällä `vkFreeCommandBuffers`-funktiota. Komentopuskurit voidaan myös nollata, joko yksitellen käyttämällä `vkResetCommandBuffer`-funktiota siinä tilanteessa, että komentopoolille on luomisvaiheessa määritetty `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT`-arvo, tai kaikki komentopoolista varatut komentopuskurit voidaan nollata käyttämällä `vkResetCommandPool`-funktiota [15.]

4.6 Datan siirtäminen näytönohjaimelle

Vulkanissa dataa siirretään näytönohjaimelle käyttämällä erilaisia puskureita. Puskureita luodaan ensimmäiseksi määrittämällä puskurien käyttötarkoitus ja tämän jälkeen varaamalla puskurin tarvitsema muisti näytönohjaimesta.

4.6.1 Verteksipuskurit

Verteksipuskurilla tarkoitetaan puskuria, joka sisältää esimerkiksi 3D-mallien verteksidataa. Verteksipuskuri on normaali Vulkanin puskuri, mutta sen käyttämiseksi kuvien piirtämisessä tulee ensin määrittää, mitä dataa se sisältää. Tämä tapahtuu luomalla verteksidatan määritys [18, s. 271–272.]

```
struct Vertex
{
    glm::vec3 position;
    glm::vec3 color;
};
```

Aluksi täytyy määrittää mitä verteksi pitää sisällään. Tässä tapauksessa verteksidata sisältää sijainti vektorin ja vektorin, joka määrittelee verteksin värin. Tämän jälkeen voidaan määrittää verteksidatan sisältö Vulkanille.

```
VkVertexInputBindingDescription vertexInputDescription = {};
vertexInputDescription.binding = 0;
vertexInputDescription.stride = sizeof(Vertex);
vertexInputDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;

std::vector<VkVertexInputAttributeDescription>
attributeDescriptions(2);

attributeDescriptions[0].binding = 0;
attributeDescriptions[0].location = 0;
attributeDescriptions[0].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[0].offset = offsetof(Vertex, position);

attributeDescriptions[1].binding = 0;
attributeDescriptions[1].location = 1;
attributeDescriptions[1].format = VK_FORMAT_R32G32B32_SFLOAT;
attributeDescriptions[1].offset = offsetof(Vertex, color);
```

Tässä määritetään aluksi verteksidatan koko käyttämällä `VkVertexInputBindingDescription`-oliota ja kuinka sitä tulisi lukea [14]. Tämän jälkeen määritetään, kuinka verteksidata liitetään varjostimeen (liite 1) käyttämällä `VkVertexInputAttributeDescription`-oliota [14].

Tämän jälkeen verteksidatan määritys tulisi lisätä grafiikkaliukuhihnan luonnissa mainittuun verteksidatan määritykseen.

Ennen verteksipuskurin luomista määritetään esimerkki dataa, joka piirtää yhden kolmion.

```
std::vector<Vertex> vertices =
{
    { { 1.0f, 1.0f, 0.0f }, { 1.0f, 0.0f, 0.0f } },
    { { -1.0f, 1.0f, 0.0f }, { 0.0f, 1.0f, 0.0f } },
    { { 0.0f, -1.0f, 0.0f}, { 0.0f, 0.0f, 1.0f } }
};
```

Tämän jälkeen voidaan luoda varsinainen verteksipuskuri.

```
VkBuffer vertexBuffer;

VkBufferCreateInfo vertexBufferCreateInfo = {};
vertexBufferCreateInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
vertexBufferCreateInfo.size = sizeof(Vertex) * vertices.size();
vertexBufferCreateInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;

vkCreateBuffer(device, &vertexBufferCreateInfo, nullptr,
&vertexBuffer);
```

Tässä määritetään puskuri käyttämällä `VkBufferCreateInfo`-oliota ja siihen asetetaan verteksidatan koko, sekä määritetään puskurin käyttötavaksi verteksipuskuri [14].

Tämän jälkeen tulee selvittää, kuinka paljon muistia puskuri tarvitsee.

```
VkMemoryRequirements memoryRequirements;
vkGetBufferMemoryRequirements(device, vertexBuffer,
&memoryRequirements);
```

Tässä puskurin vaatima muistin määrä selvitetään käyttämällä `vkGetBufferMemoryRequirements`-funktiota ja tallennetaan `VkMemoryRequirements`-olioon.

Tämän jälkeen voidaan varata puskurin tarvitsema muisti.

```
VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties(gpu, &memoryProperties);

VkMemoryPropertyFlags properties = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT
| VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;

uint32_t memoryType = 0;
for (uint32_t i = 0; i < memoryProperties.memoryTypeCount; i++)
{
    if (memoryRequirements.memoryTypeBits & 1 << i
```

```
        && (memoryProperties.memoryTypes[i].propertyFlags & properties)
== properties)
    {
        memoryType = i;
        break;
    }
}

VkMemoryAllocateInfo memoryAllocateInfo = {};
memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
memoryAllocateInfo.allocationSize = memoryRequirements.size;
memoryAllocateInfo.memoryTypeIndex = memoryType;

VkDeviceMemory memory;

vkAllocateMemory(device, &memoryAllocateInfo, nullptr, &memory);
```

Tässä aluksi selvitetään, minkä tyyppiseen muistiin puskuri voidaan varata käyttämällä `VkPhysicalDeviceMemoryProperties`-oliota ja `vkGetPhysicalDeviceMemoryProperties`-funktiota. Tämän jälkeen varataan muistia `VkDeviceMemory`-kahvaan käyttämällä `VkMemoryAllocateInfo`-oliota ja `vkAllocateMemory`-funktiota [15.]

Lopuksi verteksidata voidaan siirtää puskuriin.

```
void* data;

vkMapMemory(vulkan.device, memory, 0,
memoryAllocateInfo.allocationSize, 0, &data);
memcpy(data, vertices.data(), verticesSize);
vkUnmapMemory(vulkan.device, memory);
```

Tässä muisti liitetään pointteriin käyttämällä `vkMapMemory`-funktiota, jonka avulla verteksidata kopioidaan puskuriin, ja lopuksi puskurin liitos vapautetaan käyttämällä `vkUnmapMemory`-funktiota [14]. Tämän jälkeen verteksipuskuri on valmis käytettäväksi piirtämisessä.

Verteksipuskuri tulee tuhota manuaalisesti käyttämällä `vkDestroyBuffer`-funktiota, ja verteksipuskurille varattu muisti tulee vapauttaa manuaalisesti käyttämällä `vkFreeMemory`-funktiota [15].

4.6.2 Indeksipuskurit

Indeksipuskurilla tarkoitetaan puskuria, joka sisältää piirtämisessä käytettävää indeksidataa. Indeksidata mahdollistaa saman verteksin käyttämisen useassa kohdassa määrittämällä listan verteksien järjestysluvuista, joka määrittää missä järjestyksessä verteksit piirretään [18, s. 272–277.]

Indeksipuskurin luomiseksi määritetään ensin esimerkkidataa.

```
std::vector<uint32_t> indices = { 0, 1, 2 };
```

Tässä määritetään indeksit yksinkertaisen kolmion piirtämiseksi. Tässä voidaan huomioida, että yksittäisen kolmion piirtämiseksi ei ole mitään käytännöllistä hyötyä indeksipuskurista, mutta esimerkiksi 3D-mallin piirtäminen hyötyisi tästä huomattavasti.

Indeksipuskuri luodaan melkein samalla tavalla, kuin aiemmin mainittu verteksipuskuri. Erona on, että indeksipuskurin sisällöstä ei tarvitse tehdä määritelmää Vulkanille, ja puskuria luodessa käytetään `VK_BUFFER_USAGE_INDEX_BUFFER_BIT`-arvoa.

Kuten verteksipuskuri myös indeksipuskuri ja sille varattu muisti tulee tuhota manuaalisesti.

4.7 Kuvan piirtäminen

Vulkanissa kuvan piirtäminen tarkoittaa vaihtoketjun kuvaan grafiikan piirtämistä ja sen näyttämistä käyttäjälle. Kuvan piirtämiseksi tuleekin siis pyytää vaihtoketjulta käytettävä kuva, ja sen jälkeen komentopuskureiden avulla käyttää sitä piirtämiseen. Kuvan piirtämisen vaiheet tulee myös manuaalisesti synkronoida, jotta kuvaan ei esimerkiksi piirretä ennen kuin se on käytettävissä.

4.7.1 Semaforit

Vulkanissa piirtämisen synkronointiin voidaan käyttää esimerkiksi semaforeja [18, s. 381–384].

Semaforien luominen on hyvin yksinkertaista ja se tapahtuu seuraavasti.

```
VkSemaphore imageAvailable;

VkSemaphoreCreateInfo semaphoreCreateInfo = {};
semaphoreCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

vkCreateSemaphore(device, &semaphoreCreateInfo, nullptr,
&imageAvailable);
```

Tässä luodaan semafori käyttämällä `VkSemaphoreCreateInfo`-oliota, johon ei tarvitse määrittää mitään arvoja, ja `vkCreateSemaphore`-funktiota.

Synkronoinnin toteuttamiseksi semaforeilla tulisi myös luoda toinen semafori kuvan piirtämisen loppua varten, jotta tiedetään, milloin kuva on valmis näytettäväksi käyttäjälle. Tähän semaforiin viitataan myöhemmin nimellä `renderComplete`.

4.7.2 Komentopuskurin täyttäminen

Ennen piirtämistä tulee kaikki piirtämisessä vaaditut komennot lisätä komentopuskuriin. Tässä esimerkissä komentopuskuri ei koskaan muutu, joten kaikki tarvittavat komennot voidaan lisätä yhdellä kertaa, ja uudelleen käyttää täytettyä komentopuskuria jokaisen kuvan piirtämiseksi. Seuraavissa koodiesimerkeissä oletetaan, että kaikilla vaihtoketjun kuvilla on myös oma komentopuskuri ja koodi tapahtuu silmukan sisällä.

Komentojen lisäämiseksi täytyy ensin komentopuskurille kertoa, että komentoja aletaan syöttää.

```
VkCommandBufferBeginInfo commandBufferBeginInfo = {};
commandBufferBeginInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;

vkBeginCommandBuffer(swapChainBuffers[i].commandBuffer,
```

```
&commandBufferBeginInfo);
```

Tässä komentopuskurille kerrotaan komentojen syöttämisen alkamisesta käyttämällä `VkCommandBufferBeginInfo`-oliota ja `vkBeginCommandBuffer`-funktiota [14].

Seuraavaksi tulee määrittää piirtokierroksen alku.

```
VkClearValue clearValue;
clearValue.color = {0.0f, 0.0f, 0.0f, 1.0f};

VkRenderPassBeginInfo renderPassBeginInfo = {};
renderPassBeginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassBeginInfo.renderPass = renderPass;
renderPassBeginInfo.renderArea.offset.x = 0;
renderPassBeginInfo.renderArea.offset.y = 0;
renderPassBeginInfo.renderArea.extent.width = windowWidth;
renderPassBeginInfo.renderArea.extent.height = windowHeight;
renderPassBeginInfo.clearValueCount = 1;
renderPassBeginInfo.pClearValues = &clearValue;
renderPassBeginInfo.framebuffer = framebuffers[i];

vkCmdBeginRenderPass(swapChainBuffers[i].commandBuffer,
&renderPassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
```

Tässä määritetään `VkRenderPassBeginInfo`-olio, johon asetetaan piirtokierros, ikkunan koko, kuvapuskurit ja kuvan tyhjentämisen väri. Tämän jälkeen piirtokierros aloitetaan `vkCmdBeginRenderPass`-funktiolla [15.]

Tämän jälkeen komentopuskuriin voidaan syöttää piirtokomentoja ja ensimmäiseksi määritetään aiemmin dynaamiseksi tilaksi määritetty kuvaportin koko.

```
VkViewport viewport = {};
viewport.height = static_cast<float>(windowHeight);
viewport.width = static_cast<float>(windowWidth);
viewport.minDepth = 0.0f;
viewport.maxDepth = 1.0f;
vkCmdSetViewport(swapChainBuffers[i].commandBuffer, 0, 1, &viewport);

VkRect2D scissor = {};
scissor.extent.width = windowWidth;
scissor.extent.height = windowHeight;
scissor.offset.x = 0;
scissor.offset.y = 0;
vkCmdSetScissor(swapChainBuffers[i].commandBuffer, 0, 1, &scissor);
```

Tässä määritetään kuvaportin koko käyttämällä `VkViewport`- ja `VkRect2D`-oliota sekä käyttämällä `vkCmdSetViewport`- ja `vkCmdSetScissor`-funktiota [14].

Seuraavaksi tulee määrittää, mitä grafiikkaliukuhihnaa komentopuskuri käyttää.

```
vkCmdBindPipeline(swapChainBuffers[i].commandBuffer,
VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
```

Tässä grafiikkaliukuhihna määritetään käyttöön funktiolla `vkCmdBindPipeline` [15].

Seuraavaksi voidaan liittää verteksi- ja indeksipuskuri sekä lisätä komentopuskuriin piirtämiskomento.

```
VkDeviceSize offsets[1] = { 0 };
vkCmdBindVertexBuffers(swapChainBuffers[i].commandBuffer,
0, 1, &vertexBuffer, offsets);

vkCmdBindIndexBuffer(vulkan.swapChainBuffers[i].commandBuffer,
indexBuffer, 0, VK_INDEX_TYPE_UINT32);

vkCmdDrawIndexed(swapChainBuffers[i].commandBuffer, indices.size(), 1,
0, 0, 1);
```

Tässä käytettävä verteksipuskuri määritetään funktiolla `vkCmdBindVertexBuffers` ja käytettävä indeksipuskuri funktiolla `vkCmdBindIndexBuffer`. Tämän jälkeen komentopuskuriin lisätään indeksipuskuria käyttävä piirtokomento käyttämällä funktiota `vkCmdDrawIndexed`, mikä piirtää kuvan komentopuskuriin määritetyillä tiedoilla [14.]

Lopuksi lopetetaan piirtokierros ja komentopuskuriin komentojen lisäämisen.

```
vkCmdEndRenderPass(swapChainBuffers[i].commandBuffer);
vkEndCommandBuffer(swapChainBuffers[i].commandBuffer);
```

Tässä piirtokierros lopetetaan `vkCmdEndRenderPass`-funktiolla ja komentojen lisääminen komentopuskuriin `vkEndCommandBuffer`-funktiolla [14].

4.7.3 Kuvan esittäminen

Kun komentopuskuri on täytetty halutuilla komennoilla, voidaan se lähettää komentojonolle. Tämän jälkeen kuva esiteetään ikkunassa.

Ennen kuin komentoja voidaan lähettää eteenpäin, tulee varmistaa, että laite on valmis ja pyytää vaihtoketjulta kuva.

```
vkDeviceWaitIdle(device);

uint32_t currentBuffer;

vkAcquireNextImageKHR(device, vulkan.swapChain,
std::numeric_limits<uint64_t>::max(),semaphores.imageAvailable,
VK_NULL_HANDLE, &currentBuffer);
```

Tässä odotetaan ensin, että laite on valmis piirtämään käyttämällä `vkDeviceWaitIdle`-funktiota. Tämän jälkeen pyydetään vaihtoketjulta seuraavaa kuvaa käyttämällä `vkAcquireNextImageKHR`-funktiota [14.]

Seuraavaksi voidaan lähettää komentopuskuri komentojonolle.

```
VkPipelineStageFlags pipelineStages =
VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;

VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.pWaitDstStageMask = &pipelineStages;
submitInfo.waitSemaphoreCount = 1;
submitInfo.pWaitSemaphores = &imageAvailable;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers =
&swapChainBuffers[currentBuffer].commandBuffer;
submitInfo.signalSemaphoreCount = 1;
submitInfo.pSignalSemaphores = &renderComplete;

vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
```

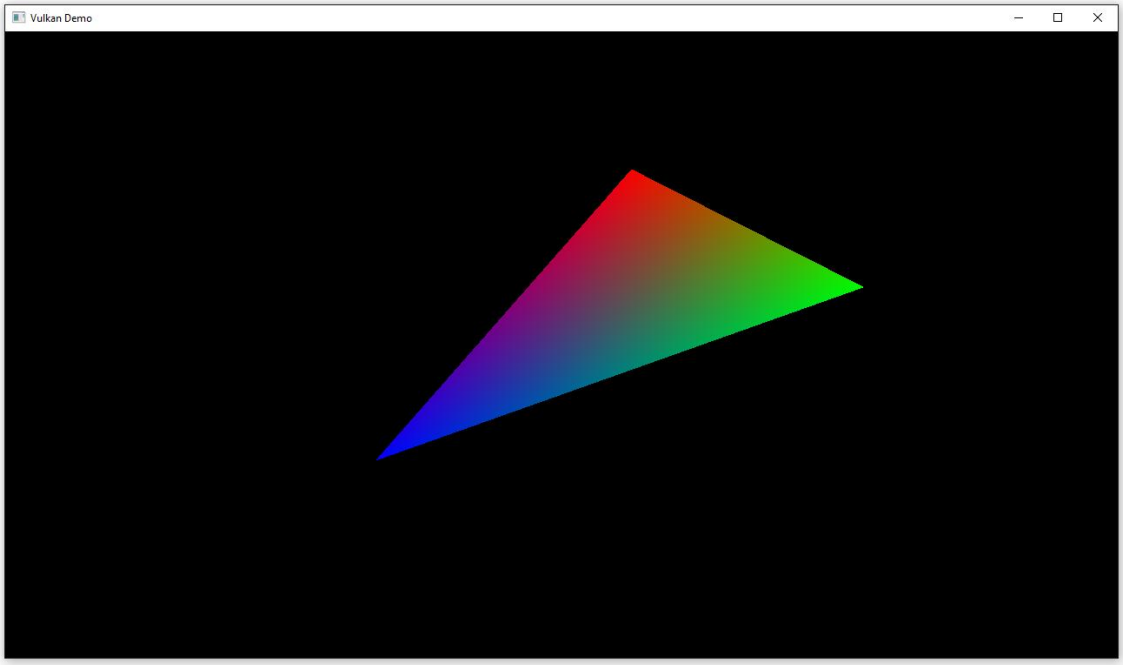
Tässä määritetään `VkSubmitInfo`-olio, johon asetetaan lähettävät komentopuskurit ja synkronoimiseen käytettävät semaforit. Tämän jälkeen komentopuskuri lähetetään komentojonolle käyttämällä `vkQueueSubmit`-funktiota [15.]

Lopuksi kuva voidaan näyttää ikkunassa.

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapChain;
presentInfo.pImageIndices = &currentBuffer;
presentInfo.waitSemaphoreCount = 1;
presentInfo.pWaitSemaphores = &renderComplete;

vkQueuePresentKHR(queue, &presentInfo);
```

Tässä määritetään `VkPresentInfoKHR`-olio, johon asetetaan käytettävä vaihtoketju ja vaihtoketjun näytettävä kuva sekä synkronoinnissa käytettävä semafori. Lopuksi kuva esitetään ikkunassa käyttämällä `vkQueuePresentKHR`-funktiota (kuva 7) [14.]



Kuva 7. Esimerkkiohjelman piirtämä kolmio.

5 YHTEENVETO

Tämän opinnäytetyön tavoitteena oli toteuttaa katsaus Vulkan-grafiikkarajapinnan perusteisiin ja sen eroihin muihin grafiikkarajapintoihin verrattuna. Lopputuloksena tuotetuilla ohjeella on mahdollista perehtyä Vulkanin toimintaan ja sen ominaisuuksiin muihin grafiikkarajapintoihin verrattuna. Tämän lisäksi sitä voidaan käyttää lähtökohtana aiheen syvempään tarkasteluun esimerkiksi pelimoottorin tuottamisen näkökulmasta.

Henkilökohtaisen tavoitteena oli opiskella käyttämään uusia moderneja grafiikkarajapintoja ja näistä erityisesti Vulkania. Mielestäni opin perusteet suhteellisen hyvin ja tavoitteenani on oppia tulevaisuudessa lisää tietoa aiheesta. Työn tekemisen aikana huomasin myös, että Vulkan eroaa suuresti vanhemmista grafiikkarajapinnoista ja sen uutuudesta johtuen siitä oli vaikea löytää tietoa.

Vulkan mahdollistaa näytönohjaimen resurssien tehokkaamman hyödyntämisen siirtämällä näytönohjaimen hallintaan vaadittavaa koodia laiteajurista ohjelmaan. Tämä nostaa koodin monimutkaisuuttaa ja pakottaa ohjelman määrittämään kaikki vaaditut omanaisuudet selkeästi. Kuitenkin tästä johtuen, koska ohjelma voi täsmällisestä määrittää vaatimansa resurssit ja ominaisuudet, on mahdollista käyttää nykytietokoneiden tarjoamia resursseja huomattavasti optimaalisemmin, kuin aiemmat grafiikkarajapinnat ovat mahdollistaneet.

LÄHTEET

WWW-julkaisuihin viitattu 24.04.2017

1. Sgi: OpenGL 2.0 Unleashes the Power of Programmable Shaders. [WWW-Julkaisu] <https://web.archive.org/web/20040822002445/http://www.sgi.com/company_info/newsroom/press_releases/2004/august/opengl.html>

2. Eisler Craig: DirectX Then and Now (Part 1). [WWW-Julkaisu] <http://craig.theeislers.com/2006/02/20/directx-then-and-now-part-1/>

3. Khronos Group: Vulkan™ Overview. [WWW-Julkaisu] <https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf>

4. Khronos Group: OpenGL ARB to Pass Control of OpenGL Specification to Khronos Group. [WWW-Julkaisu] <https://www.khronos.org/news/press/opengl_arb_to_pass_control_of_opengl_specification_to_khronos_group>

5. Microsoft: What is Direct3D 12? [WWW-Julkaisu] <https://msdn.microsoft.com/en-us/library/windows/desktop/dn899228(v=vs.85).aspx>

6. Forbes: AMD and DICE To Co-Develop Console Style API For Radeon Graphics. [WWW-Julkaisu] <https://www.forbes.com/sites/davealtavilla/2013/09/30/amd-and-dice-to-co-develop-console-style-api-for-radeon-graphics>

7. AMD: One of Mantle's Futures: Vulkan. [WWW-Julkaisu] <https://community.amd.com/community/gaming/blog/2015/05/12/one-of-mantles-futures-vulkan>

8. Khronos Group: Vulkan. [WWW-Julkaisu] <https://www.khronos.org/vulkan/>

9. Anandtech: Khronos Announces Next Generation OpenGL Initiative. [WWW-Julkaisu] <http://www.anandtech.com/show/8363/khronos-announces-next-generation-opengl-initiative>

10. Develop: glNext revealed as Vulkan graphics API. [WWW-Julkaisu] <https://www.develop-online.net/news/glnext-revealed-as-vulkan-graphics-api/0203867>

11. Khronos Group: Khronos Releases Vulkan 1.0 Specification. [WWW-Julkaisu] <https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification>

12. GLFW: GLFW. [WWW-Julkaisu] <http://www.glfw.org/>

13. LunarG: Vulkan SDK. [WWW-Julkaisu] <https://vulkan.lunarg.com/>

14. Khronos Group: Vulkan® 1.0.47 - A Specification (with all registered Vulkan extensions). [WWW-Julkaisu] <https://www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html>

15. Khronos Group: Vulkan API Reference Pages. [WWW-Julkaisu] <https://www.khronos.org/registry/vulkan/specs/1.0/apispec.html>

16. GLFW: Vulkan guide. [WWW-Julkaisu] <http://www.glfw.org/docs/latest/vulkan_guide.html>

17. LunarG: Vulkan Validation and Debugging Layers. [WWW-Julkaisu] <https://vulkan.lunarg.com/doc/sdk/1.0.46.0/windows/layers.html>

18. Sellers Graham, Kessenich John: Vulkan Programming Guide: The Official Guide to Learning Vulkan. Addison-Wesley 2016. ISBN: 978-0-13-446454-1

19. Bomfim Diney: Khronos EGL and Apple EAGL. [WWW-Julkaisu] <http://blog.db-in.com/khronos-egl-and-apple-eagl/>

20. Vivo Patricio Gonzalez, Lowe Jen: What is a fragment shader? [WWW-Julkaisu] <https://thebookofshaders.com/01/>

21. LunarG: SPIR-V Toolchain. [WWW-Julkaisu] <https://vulkan.lunarg.com/doc/sdk/1.0.46.0/windows/spirv_toolchain.html>

LIITTEET

LIITE 1. SIMPLEVERTEXSHADER.VERT

```
#version 450 core

#extension GL_ARB_separate_shader_objects : enable

layout (location = 0) in vec3 inPos;
layout (location = 1) in vec3 inColor;

layout (location = 0) out vec3 outColor;

void main()
{
	outColor = inColor;
	gl_Position = vec4(inPos.xyz, 1.0);
}
```

LIITE 2. SIMPLEFRAGMENTSHADER.FRAG

```
#version 450 core

#extension GL_ARB_separate_shader_objects : enable

layout (location = 0) in vec3 color;

layout (location = 0) out vec4 outFragColor;

void main()
{
    outFragColor = vec4(color, 1.0);
}
```